SPR: Shielded Processor Reservations with Bounded Management Overhead

Esma Kökten[1] Gabriel Parmer[1] Björn B. Brandenburg[2]

[1]The George Washington University, Washington, DC
[2]Max Planck Institute for Software Systems, Kaiserslautern, Germany

***Abstract*—With growing hardware consolidation in modern computational infrastructures, ensuring predictable CPU allocation has become increasingly critical. Processor reservations, usually realized through rate-limiting servers, play an essential role in providing such predictability by precisely controlling when and how long each task may execute. In theory, rate-limiting servers provide strong temporal isolation, meaning that a task's timely access to its guaranteed budget is not contingent on the behavior of any other task in the system. However, in practice, these guarantees are easily undermined by the realities of actual hardware and shortcomings in naïve reservation implementations. When confronted with malicious tasks using the reservation policy itself to attack the very security and rate properties it is meant to uphold, temporal isolation breaks down in current implementations. In response, this paper presents *shielded processor reservation* (SPR) scheduling, a novel approach that ensures that at most two reservations are processed per scheduler invocation and integrates deferred timer handling, early replenishment processing, and processor access granularity guarantees to provide robust temporal isolation. We implement SPR and existing rate-limiting servers in the Composite operating system and evaluate its performance. The results demonstrate that SPR provides reliable rate-limiting with low overhead while also mitigating vulnerabilities that can be exploited to attack existing reservation systems.**

I. Introduction

Rate-limiting servers [2, 20, 22] are a core mechanism in our computational infrastructure. Widely used in Linux [12], in hypervisors such as Xen [24], and at the core of CPU control in `cgroups` containers [1], they restrict CPU usage to provide *virtual processor shares* that give tenants predictable access to processing capacity. As such, they serve as a critical security measure to maintain desired levels of availability.

For example, *aperiodic* tasks—*i.e.,* tasks without an inherent limit on the rate of activations such as network interrupts—pose a risk of excessive delays and can, in the worst case, even cause livelock [16]. More generally, untrusted tasks, aperiodic tasks exposed to unpredictable inputs, and any other potentially unbounded workloads, must be *temporally isolated* to prevent them from unduly interfering with critical time-sensitive tasks.

Starting with RT-Mach [14] and popularized by the Resource Kernel [17], the established solution to this problem is to encapsulate such tasks in *reservations* backed by rate-limiting servers that shape aperiodic (or overrunning) tasks into a predictable pattern of interference, which enables their integration into traditional schedulability analysis. A rate-limiting server is typically defined in terms of a task τ_i associated with a *budget* b_i that is *consumed* while τ_i executes. A *replenishment rule* determines when the budget is increased. For example, a *deferrable server* [22] replenishes the budget to its initial value at the beginning of every *period* p_i, thus limiting τ_i's asymptotic utilization to b_i/p_i.

Critically, reservations are used in practice not only to contain aperiodic tasks that may "accidentally" cause excessive interference (*e.g.,* due to input overload or other benign factors), but also in security-centric systems where actively malicious behavior on behalf of the encapsulated tasks cannot be ruled out. For example, seL4 [8] integrates *sporadic servers* [13, 20] into the execution and IPC subsystems to constrain untrusted processes, and TCaps [9] combine constrained budgets and a mechanism to transfer budget slices alongside invocations to predictably coordinate among schedulers.

Tenants that actively try to undermine the temporal isolation guarantee of the reservation system pose new challenges. While reservations are very well understood from a schedulability perspective, their practical implementation has received far less attention, which is unfortunate as it is decidedly nontrivial to implement reservations correctly (*i.e.,* such that their theoretical isolation guarantees are realized in practice).

Already in 2010, Stanovich et al. [21] uncovered flaws in the POSIX sporadic server specification and demonstrated that tasks can consume processor capacity beyond the intended limits. Much more recently, Mergendahl et al. [15] demonstrated *Thundering Herd* attacks against the sporadic server implementation in seL4 [13]. In particular, they showed that a victim thread's temporal isolation guarantee can be violated by inducing unanticipated interference (*i.e.,* priority inversion that *theoretically* should be impossible) that is bounded only in the number of attacker-controlled threads. Interestingly, in this case, the very security mechanism designed to provide predictable reservations opened the door to an attack that negated the desired temporal isolation guarantees.

This paper. Motivated by these findings, we revisit the issue of reservations for untrusted, potentially malicious tasks in light of the realities of actual hardware and the shortcomings of naïve implementations, and systematically investigate how rate-limiting servers must be implemented to realize their temporal isolation guarantees in practice. We identify three main attack vectors that defeat temporal isolation, against which no published reservation scheme, and to the best of our knowledge, no publicly available implementation, defends.

A1 Lower-priority tasks can coordinate to force the processing of N reservation replenishments on the critical path to dispatching a higher-priority task, where N is the number of attacker-controlled tasks. The accumulated replenishment processing can induce excessive latency and ultimately, for large N, a breakdown of temporal isolation. Mergendahl et al.'s Thundering Herd [15] falls into this category.

A2 Lower-priority tasks can arrange for the processing of a large number of slightly offset replenishment timers during the execution of a higher-priority task, resulting in excessive interrupt delays, with cumulative effects similar to A1.

A3 Higher-priority tasks can cause an unbounded number of virtually back-to-back preemptions in lower-priority tasks, causing excessive context switching overhead and preventing lower-priority tasks from effectively using their guaranteed processor time.

The two *low-on-high* attacks A1 and A2 exploit the fact that untrusted tasks can tailor their behavior to trick naïve server implementations into performing management operations at inopportune times: they *use the reservation policy itself to attack the very isolation properties it is meant to uphold.*

The core issue underlying the *high-on-low* attack A3 is that prior work has reasoned only about (unqualified) processor time, but not made guarantees on *useful* processing time. The two are not the same on real hardware, since context switching is not free and disturbs the cache state, and budget tracking has accuracy limits. Processor time is a fungible resource only on paper; what is needed in reality is *progress guarantees*.

This paper presents our solution, *shielded processor reservation* (or SPR) scheduling, which categorically defends against all three attack vectors. SPR is based on three novel principles not previously articulated in the context of reservation systems:

- A budget replenishment that does not cause a task to be dispatched *immediately* can be safely shifted in time.
- A timer interrupt that does not cause an *immediate* context switch is unnecessary and can be safely omitted.
- A meaningful processing time guarantee must extend to the *granularity of allocation*, not just its cumulative duration.

Informed by these principles, SPR combines *early replenishment processing*, *deferred timer processing*, and scheduling logic that processes *at most two reservations* per invocation, which thwarts attacks A1 and A2. Additionally, SPR allows for *non-preemptive regions* as part of its reservation contracts (accounted for in admission control), which addresses A3.

Contributions.

- We demonstrate attacks A1–A3 on existing sporadic and deferrable server implementations, and show the induced overhead to lead to a breakdown in temporal isolation.
- From the attacks, we derive novel, not previously articulated design principles for reservation implementations.
- We introduce SPR scheduling, which ensures bounded reservation management overhead on the critical path.
- We describe how SPR scheduling integrates non-preemptive regions to give processor access granularity guarantees.
- We implement and evaluate SPR scheduling in the Composite OS and demonstrate that predictable reservation processing is possible, with very low overhead.

II. Model and Assumptions

In this paper, we focus on fixed-priority scheduling, though the SPR principles transfer to other settings, including EDF scheduling and non-priority-based scheduling (*e.g.,* round-robin or FCFS). For simplicity, we restrict our focus to single-core scheduling and partitioned multiprocessor scheduling.

Tasks. The system consists of a set of *tasks* $\{\tau_1, \tau_2, \ldots\} = \mathcal{T}$ (or, equivalently, $|\mathcal{T}|$ *servers* containing one task each). After a task *activates*, it remains *ready* until *completion*, at which point it becomes *inactive* again and awaits its next activation. While it is ready, a task is either executing or preempted.

As we focus on reservations, we make few assumptions about the underlying real-time task model. In particular, we allow for an aperiodic task model in which each task's inter-activation time is unrestricted. This admits both classical periodic and sporadic tasks with bounded inter-activation times, but also general event-driven tasks such as interrupt handlers.

Reservations. Reservation systems define budget consumption, depletion, and replenishment rules. We consider the popular deferrable [22] and sporadic server [20] policies. Both schemes describe each task τ_i with a fixed priority σ_i, a *budget* b_i, a *current budget* $\hat{b}_i$, and a *replenishment period* p_i. The *utilization* b_i/p_i defines the reserved share of the processor.

Initially, $\hat{b}_i = b_i$. When task τ_i executes, its current budget $\hat{b}_i$ decreases at unit rate under both policies. When $\hat{b}_i = 0$, the current budget is *depleted*, which signals that the allowed rate of execution has been reached, and τ_i is removed from the runqueue until the current budget is *replenished.*

Deferrable and sporadic servers differ in how and when a task's current budget is replenished. Deferrable servers reset $\hat{b}_i$ to b_i periodically every p_i, at times divisible by p_i. As such, deferrable servers replenish at fixed times irrespective of actual budget use—leading to the possibility of a "double hit" of $2 \times b_i$ contiguous consumption, once at the end of one period window and continuing at the start of the next.

In contrast, sporadic servers track budget consumption precisely: if a task τ_i is activated at time t_s and executes for e time units before becoming inactive again or depleting its current budget at time t_e, then e is added to τ_i's current budget $\hat{b}_i$ at time $t_s + p_i$. Theoretically, a sporadic server can generate a (virtually) unbounded number of pending replenishments. In practice, implementations often impose a limit on the number of replenishments tracked for each reservation to bound memory consumption [13, 21]. We thus assume that each sporadic server's number of pending replenishments is bounded by a (configurable) constant.

A policy-specific *admission test* such as response-time analysis is used to ensure that the set of servers $\mathcal{T}$ is schedulable, which prevents overload and ensures that every server can consume its entire budget within every replenishment period, irrespective of whether (or when) other servers are executing.

Timers. The OS uses timer interrupts to keep track of time and to react to time-sensitive events. These events include budget depletion, replenishment handling, and task activation after time-based self-suspension operations (*e.g.,* `nanosleep`).

Timer granularity can significantly complicate reservation accounting. Traditionally, many real-time systems relied on

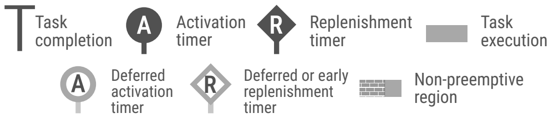


Fig. 1: Legend for all examples.

periodic "timer ticks," *i.e.,* periodic timer interrupts that trigger every *quantum*. However, such tick-based systems suffer from a potential delay between when events are intended to trigger and when the next subsequent timer interrupt actually fires. The resulting uncertainty in time-keeping causes notable complications for budget tracking and enforcement [21].

In contrast, we assume a "tickless" system in which timer interrupts can be programmed at the granularity of cycles to occur exactly at the time of the next expiring timer. Such "one-shot" timers are widely available in modern hardware.

III. Design

This section outlines the design of SPR and illustrates how it overcomes the limitations identified in existing reservation systems discussed in §I. We focus on deferrable and sporadic servers due to their popularity, though we believe that the observations and approaches apply generally. We first illustrate unnecessary timer interrupts (§III-A).To address these, SPR employs the following three techniques: 1) *early replenishments* (§III-B), 2) *pruning* of premature activation and replenishment timers (§III-C), and 3) *deferred* activation and replenishment processing (§III-D). Additionally, as previously demonstrated by the Thundering Herd attack [15], the *eager* processing of a potentially unbounded number of reservations during a single invocation of the scheduler or timer logic exposes an attack surface. To counteract this weakness, we introduce *prioritized, bounded reservation processing* (§III-E). Lastly, we integrate *non-preemptive regions* (§III-F) to ensure spans of cache-friendly contiguous computation.

Throughout the rest of the paper, we provide concrete examples to illustrate the key issues using three tasks τ_{hi}, τ_{mid}, and τ_{lo} at specific priority bands $\sigma_{hi} > \sigma_{mid} > \sigma_{lo}$. Table I lists the replenishment periods, budgets, and priorities of the tasks. The legend for all examples is given in Figure 1.

A. Unnecessary Timer Interrupts

Figure 2 illustrates multiple scenarios in which unnecessary timer interrupts can occur in a naïve reservation implementation. *Unnecessary* timer interrupts are those that update bookkeeping information (*e.g.,* for reservations), but do not result in a context switch between tasks (*i.e.,* that do not alter the task schedule). Tasks are placed on the Y-axis, and the X-axis represents time. Three lines represent the budgets of the tasks, and the shaded areas represent the execution of the tasks. The budget lines illustrate how budget is consumed and replenished. When the line indicating the budget level reaches the "bottom" (w.r.t. each task), the respective task's budget is depleted. Circled alphabetic labels correspond to different types of unnecessary timer interrupts, as discussed next.

a *Replenishment timer for the currently executing task.* Task τ_{mid} has 10 units of budget remaining, and is scheduled for execution at time t_1. τ_{mid} is interrupted by a replenishment timer at time t_2 for a replenishment of 10 units, but a context switch does not occur. Early replenishments (§III-B) eliminate this unnecessary replenishment timer interrupt.

TABLE I: Tasks used in the examples.

Task (τ)	Period (p)	Budget (b)	Priority (σ)
τ_{hi}	30	10	σ_{hi}
τ_{mid}	40	20	σ_{mid}
τ_{lo}	60	10	σ_{lo}

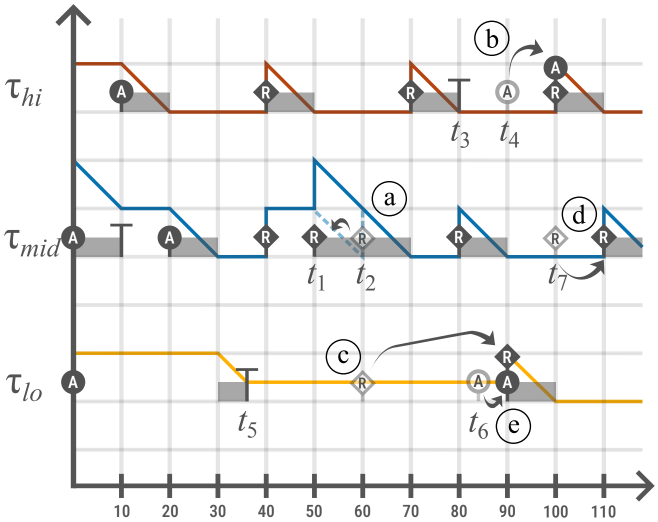


Fig. 2: Unnecessary timers caused by sporadic servers.

b *Activation timer in a depleted state.* Task τ_{hi} completes execution at time t_3 and depletes its budget while transitioning to the inactive state. Note that a task can both complete and become depleted as completion is typically not atomic and requires processing. Task τ_{hi} has an activation timer at time t_4, however, it has no budget, therefore it cannot be scheduled before its replenishment. In general, timer interrupts to activate depleted tasks can be delayed (§III-C).

c *Replenishment timer for a completed task.* Task τ_{lo} completes at time t_5 and has an activation at t_6. A replenishment is scheduled for time 60, in between t_5 and t_6. The resulting timer interrupt is unnecessary since it does not change the task interleaving compared to a scenario in which we delay the replenishment to time t_6, thus motivating delayed replenishment processing (§III-C).

d e *Replenishment or activation of a lower-priority task while a higher-priority task is executing.* Task τ_{mid} has its budget replenished at time t_7 when τ_{hi} starts executing. Thus, τ_{mid} is not actually scheduled until τ_{hi} depletes its budget. Similarly, τ_{lo} has an activation timer at time t_6, while τ_{mid} is executing. Although τ_{lo} is activated at time t_6, it is not scheduled until τ_{mid} depletes its budget. Generally, while a higher-priority task is executing, any timers for lower-priority tasks are unnecessary and can be delayed until all higher-priority tasks become inactive (§III-D).

While each of these forms of unnecessary timer interference may seem somewhat innocuous on their own, we show in §V that they can be *maliciously* used to attack the reservation system. To prevent all of these forms of unnecessary timer interference, SPR uses the techniques described in the following.

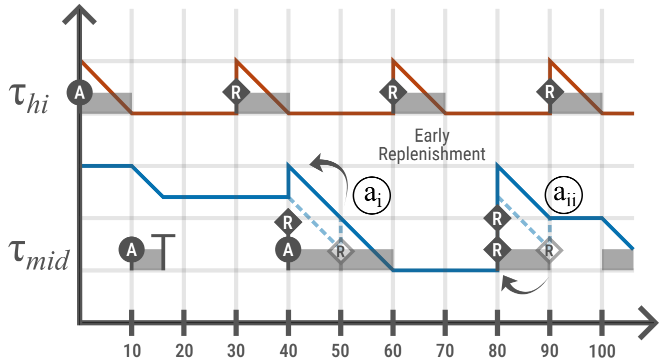


Fig. 3: Example scenario with early replenishment.

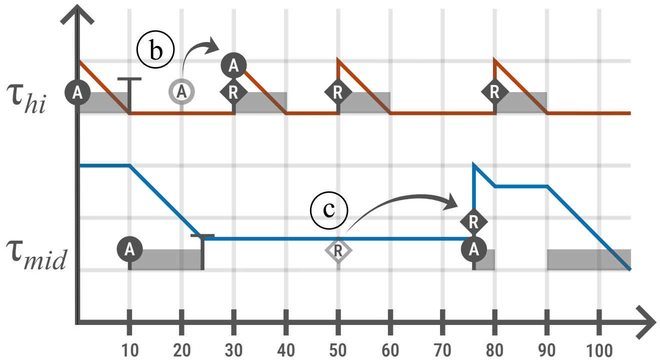


Fig. 4: Unnecessary activation timer example.

B. Early Replenishments

Replenishments for a task might be scheduled for a future time that happens to be *during* the task's execution. This would cause a timer interrupt that would not switch tasks, instead simply updating the budget, and continuing to execute the already scheduled task. Our early replenishment mechanism leverages that we know when the currently available budget overlaps with the replenishment, thus we process that replenishment *early*, just before the task is actually scheduled.

In Figure 3, the replenishment for τ_{mid} is processed early, just before its execution. The dotted lines in the budget graph represent the budget without early replenishment. As shown, the early replenishment eliminates the need for an unnecessary timer interrupt that would delay the task (a_i in Figure 3).

Note that early replenishment does *not* change the times of future replenishments. For example, the budget originally scheduled to be replenished at time 50 that is replenished early at time 40 (and consumed throughout $[50, 60)$) does not become available again until time $50 + p_{mid} = 90$.

However, as illustrated (a_ii in Figure 3), the replenishment scheduled for time 90 is again eligible for early replenishment at time 80 when τ_{mid} is dispatched. This time, though, τ_{mid} is preempted by τ_{hi}, so that τ_{mid} cannot consume its early-replenished budget until time 100, at which point τ_{mid}'s budget would have been replenished anyway. This example shows that early replenishment is orthogonal to preemptions and that it does not negatively affect higher- or lower-priority tasks.

As SPR performs early replenishments, the task's budget depletion time is pushed into the future. This postponement can result in overlaps with further replenishments, enabling further early replenishments. Ultimately, the overhead induced by early replenishments is bounded in the number of replenishments per reservation, which is typically bounded by a small constant in practical reservation schemes (including sporadic servers [21]), ensuring that the dispatch operation remains fast.

Early replenishment is safe because it applies only if the increase of a running task's budget is a foregone conclusion. Early replenishment reduces timer self-interference caused by a task's own replenishment. Additionally, the techniques described in §III-C and §III-D further delay the processing of replenishments in other scenarios, promoting additional opportunities for early replenishment.

C. Premature Activation and Replenishment Timer Pruning

On the one hand, tasks may deplete their budget while completing their execution. In such cases, a task can request an activation to take place sooner than its budget will be replenished, resulting in a premature activation timer. For example, Figure 4 shows how a premature activation timer may fire while task τ_{hi} is still depleted. By pruning the premature activation timer and delaying the task's activation until its budget is replenished (b in Figure 4), SPR ensures that the task resumes execution at the earliest time possible while eliding a source of interference for other tasks.

On the other hand, tasks may complete their execution and become inactive after partially consuming their budget. While being inactive, any replenishments for the task are obviously unnecessary, as the task is not runnable irrespective of the amount of budget available to it (c in Figure 4). As such premature replenishment timer interrupts cannot result in a context switch, SPR prunes them and processes any pending replenishments at the time of the task's next activation.

These two techniques prevent unnecessary interference due to premature replenishment and activation timers.

D. Deferred Replenishment and Activation Processing

The timer interrupt logic for processing replenishments can cause significant priority inversion. A task's replenishment or activation timer may fire while a higher-priority task is executing, regardless the priority of the task that is to be replenished or activated. In Figure 5, d represents a replenishment timer and e represents an activation timer for τ_{mid} firing while τ_{hi} is executing. These timers interfere with the execution of τ_{hi} unnecessarily, and they should be delayed until the end of τ_{hi}'s execution. To eliminate the interference caused by the timers at d and e, SPR ensures a priority-aware processing of replenishments and activations.

In general, all timeouts should be processed with the priority of their associated task's priority. Therefore, SPR does not allow replenishment or activation timers for lower-priority tasks to fire while a higher-priority task is executing, so that the lower-priority timers are safely delayed until the higher-priority task completes or depletes it budget. Note that TimerShield [18] similarly delays timers for task activations, and we extend this approach to replenishment processing.

Summary. SPR allows only timer interrupts that are necessary in the sense that they *immediately result in a context switch.*

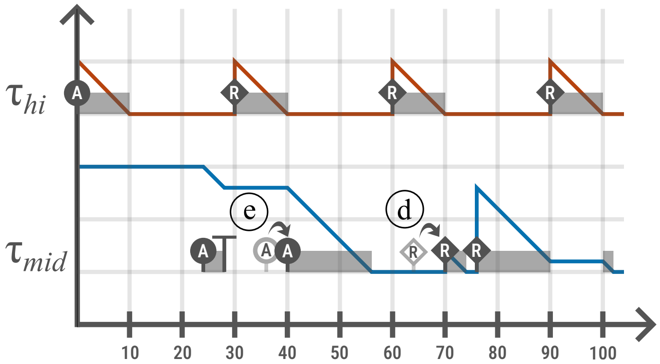


Fig. 5: Deferred timers of lower-priority tasks.

More precisely, SPR suppresses unnecessary timer interference present in previous reservation implementations by processing inevitable replenishments early (§III-B), by pruning premature activation and replenishment timers (§III-C), and by deferring lower-priority activation and replenishment timers (§III-D). This 1) can increase system throughput by decreasing the number of timer interrupts and their associated overheads, 2) ensures system predictability even when the number of threads is not known a-priori (*e.g.,* in open real-time systems), and 3) protects the system by preventing the reservation system itself becoming the focus of malicious attacks.

E. Bounded Reservation Processing in the Scheduler

Another source of potentially prolonged priority inversion occurs when many reservations are processed as part of a single scheduler invocation or timer interrupt. This may happen naturally because multiple timeouts—replenishments and activations—for different tasks may expire (and thus become due for processing) within the same timer interrupt if they happen to expire at the same (or almost the same) time. Existing sporadic server implementations handle these timeouts in a single scheduler (or timer subsystem) activation, introducing unbounded and unpredictable interference [15]. This problem is exacerbated in systems that use quantum-based timers as all timeouts that expire within a quantum are handled together at the next quantum boundary.

Techniques for early and deferred replenishment processing ensure that only necessary timer interrupts occur, but do *not* prevent a single timer interrupt from handling a large—and in open systems potentially unbounded—number of replenishments. In fact, they might *increase* the number of replenishments processed at a single time because their very purpose is to shift unnecessary replenishment processing to times at which the scheduler is called anyway.

To bound the overhead caused by reservation processing in a single scheduler invocation (or timer interrupt), SPR processes *only the timeout (i.e., replenishment or activation) of the next task to be executed*. This priority-aware processing of reservations effectively *defers* the processing of replenishments and activations of lower-priority (or same-priority) tasks until they have a chance to be scheduled right away.

Due to deferred replenishment processing, a task can accumulate multiple expired replenishments by the time it becomes the highest-priority task, at which point SPR processes all outstanding replenishments before the task is dispatched. As we assume that the number of replenishments for each task is bounded (which is typical in practical reservation implementations [13, 21]), this is a constant-time operation.

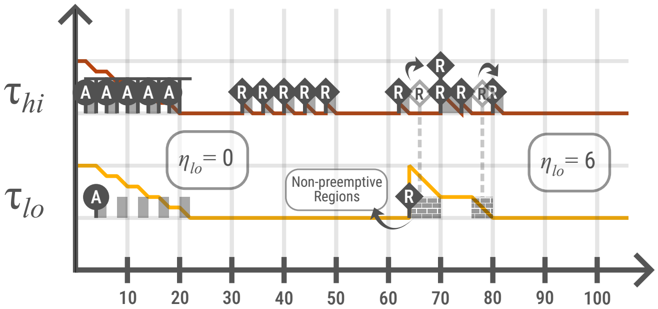


Fig. 6: Illustration of non-preemptive regions.

F. Preventing Excessive Preemption Rates

While existing reservation algorithms ensure that tasks receive processor time according to the specified rate of execution, they make no guarantees about the granularity of allocation. In particular, sporadic and deferrable servers (that do not execute with the maximum priority) provide no upper bounds on the number of preemptions incurred per server period, and hence also not on the total amount of preemption-induced overheads. That is, existing reservation schemes provide guarantees on the time that a task is *scheduled*, but cannot guarantee that tasks will make a predictable amount of computational *progress* while using their budget. As progress is often correlated with effective cache usage, we wish to decrease cache-related preemption delays.

To this end, SPR augments the classic reservation interface (b_i, p_i, σ_i) with a fourth parameter: the *non-preemptive region length* η_i. Whenever a task τ_i is dispatched (*i.e.,* chosen for execution after a context switch), SPR ensures that τ_i will not be involuntarily preempted within $\min(\eta_i, \hat{b}_i)$ time units of starting execution by deferring any higher-priority activations and replenishments until the end of the non-preemptive region. As a result, if $\eta_i > 0$, then τ_i is guaranteed to suffer at most $\left\lceil \frac{b_i}{\eta_i} \right\rceil - 1$ involuntary preemptions per server period.

The non-preemptive region mechanism enables lower-priority tasks to effectively use the cache for a guaranteed minimum amount of time, irrespective of the self-suspension behavior and activation pattern of any higher-priority tasks. The tradeoff is that it induces a bounded amount of priority inversion, which must be taken into account by the schedulability test during admission control (discussed in §III-G below).

Figure 6 illustrates a case where τ_{lo} suffers from frequent preemptions due to τ_{hi}'s activation pattern. In this example, the left-hand side of the figure shows τ_{lo}'s execution with $\eta_{lo} = 0$ subject to frequent disruptions. In contrast, the right-hand side depicts η_{lo} set to 6, which ensures that τ_{lo} can consume a minimum of 6 units of budget before being preempted. In this case, τ_{lo} is preempted only $\left\lceil \frac{10}{6} \right\rceil - 1 = 1$ times, and thus can consume its budget in two contiguous regions, which is generally beneficial for cache usage.

G. Admission Control and Overhead Analysis

Recall from §II that the goal of admission control is to ensure that all servers are schedulable, which means that every server $\tau_i \in \mathcal{T}$ has the opportunity to consume its entire budget b_i within each server period p_i, irrespective of the behavior of any other servers and assuming τ_i is continuously active. Since SPR integrates non-preemptive regions, which cause priority inversion, we adopt established response-time analysis for limited-preemptive fixed-priority scheduling [5, 6] as the admission test, which we briefly summarize in the following.

Let $I_i = \max(\{\eta_l \mid \sigma_l < \sigma_i\} \cup \{0\})$ denote the maximum lower-priority non-preemptive region length, and let, for each higher-priority server τ_h, $RBF_h(\Delta) = \alpha_h(\Delta) \times b_h$ denote τ_h's request-bound function, where $\alpha_i(\Delta) = \lceil \Delta/p_i \rceil$ if τ_h is a sporadic server, and $\alpha_i(\Delta) = \lceil (\Delta + (p_i - b_i))/p_i \rceil$ if it is a deferrable server (to account for the "double hit").

The set of servers $\mathcal{T}$ is schedulable if, for each $\tau_i \in \mathcal{T}$, there exists a bound $R_i \leq p_i$ such that[1]

$$I_i + b_i + \textstyle\sum_{\sigma_h \geq \sigma_i} RBF_h(R_i) \leq R_i. \tag{1}$$

Eq. (1) assumes that all scheduling-related overhead is charged against the current budgets of reservations involved in a context switch (*i.e.,* the time taken by the scheduler is not left unaccounted for). Therefore, let us analyze L_i, the amount of budget "lost" to overheads in the worst case (*i.e.,* the amount of budget that would need to be over-provisioned to compensate for worst-case overheads), again assuming that the task under analysis τ_i is continuously active.

To this end, let O_i^{pre} denote an upper bound on the total overhead incurred by τ_i due to one preemption, where O_i^{pre} accounts for one timer interrupt, one scheduler invocation (including any deferred replenishment and activation processing), and one context switch. Then τ_i "loses" at most

$$L_i = \left(\left\lceil \frac{b_i}{\eta_i} \right\rceil - 1 + 1 \right) \times O_i^{pre} = \left\lceil \frac{b_i}{\eta_i} \right\rceil \times O_i^{pre} \tag{2}$$

units of budget to scheduling overheads per server period, where the "+1" accounts for the initial context switch when τ_i is first scheduled in each server period. The maximum cache-related preemption delay can be bounded analogously.

In comparison with prior work, Eq. (2) reflects two key innovations of SPR. First of all, the mechanisms established in §III-B through §III-E ensure that O_i^{pre} can be reasonably bounded at all. In particular, in prior implementations, O_i^{pre} would have to account for a *potentially unbounded* number of activations and replenishments as part of a single scheduler invocation [15], which clearly precludes an effective bound.

Second, the non-preemptive region support introduced in §III-F is necessary for L_i to be reasonably bounded at all. Without the bound on the maximum number of preemptions per server period induced by η_i, Eq. (2) would have to account for a *potentially unbounded* number of preemptions since the number of preemptions would be controlled entirely by the (potentially malicious) activation patterns of higher-priority tasks, which again prevents any meaningful analysis.

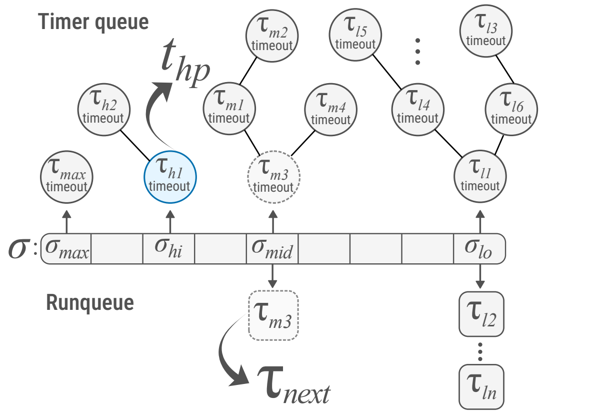


Fig. 7: Timer queue and runqueue per priority (σ). In this example, $\sigma_{max} > \sigma_{hi} > \sigma_{mid} > \sigma_{lo}$ and $\tau_{l1}.timeout < \tau_{m3}.timeout < t_{now} < \tau_{h1}.timeout < \tau_{max}.timeout$.

IV. Implementation

SPR integrates the sporadic server algorithm [20], in alignment with the guidelines provided by Stanovich et al. [21], with deferred timer handling, early replenishment processing, bounded reservation management, and non-preemptive regions, as introduced in Section III. Our implementation depends on widely available, cycle-accurate, one-shot timers to eliminate inaccuracies associated with quantum-based timers [21]. For simplicity, we present the implementation of SPR on a single core (*i.e.,* a uniprocessor or one core under partitioned multiprocessor scheduling). We implemented SPR in the Composite OS, which is a publicly available [7] μ-kernel focused on component-based system construction [23] that defines scheduling policies in user-level, isolated components. However, SPR requires no specific features of Composite; the blueprint presented here readily transfers to other systems.

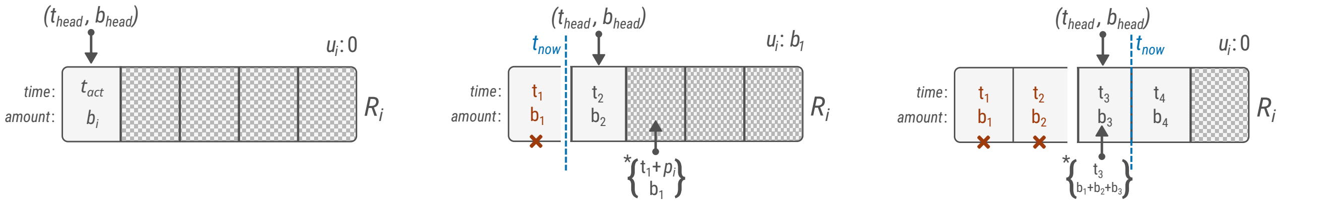
A. Data Structures

SPR relies on the following data structures.

Runqueue. As is typical in fixed-priority scheduling, we use a priority-based runqueue that stores the tasks ready for execution as an array of per-priority lists (Figure 7); possible generalizations to other runqueue designs are discussed in §V-D. The highest-priority task in the runqueue is identified in $O(1)$ time (as there are a constant number of priorities). Upon activation or replenishment, a task is moved to the runqueue; conversely, when a task completes its execution or depletes its budget, it is removed from the runqueue.

Timer queues. For each priority level, SPR maintains a separate timer queue, realized as a min-heap, as shown in Figure 7. Each node in each of the per-priority heaps stores a task's next *timeout*, denoted $\tau_i.timeout$, where the timeout is either the time of the task's next replenishment (if its budget is currently depleted) or the time of the task's next activation (if it self-suspended, *e.g.,* via `nanosleep`). This approach makes it straightforward to retrieve both the soonest timeout for a task with a priority higher than the currently active task, and the highest priority level at which there is either an active task or an expired timeout. We err toward simplicity in

[1] A tighter analysis is possible by exploiting that τ_i cannot be preempted in its final non-preemptive region [5, 6], but we prefer Eq. (1) here for simplicity.



(a) Initial replenishment queue for τ_i.

(b) A new replenishment is queued.

(c) Merging expired entries upon activation.

Fig. 8: Replenishment queue state transitions. *{}: newly added or updated values. ✖: removed entry.

our implementation, which is linear in the (constant) number of priorities, but note that Segment Trees [19] as used in TimerShield [18] could be used to realize next-timeout lookups in logarithmic time (w.r.t. the number of priorities). In the following, $timer\_queue.update(\tau_i)$ adds or updates a task τ_i's timeout value, and $timer\_queue.remove(\tau_i)$ removes the node for task τ_i from the timer queue (if currently queued).

Per-task data. In addition to a task's next timeout $\tau_i.timeout$ and its *static* parameters—its priority σ_i, budget b_i, replenishment period p_i, and non-preemptive region length η_i—SPR tracks for each task τ_i the following *dynamic* information: its *state* $\in \{ready, blocked, depleted\}$, the *used budget* u_i (so that $\hat{b}_i = b_i - u_i$), the last *execution start time* x_i (*i.e.,* time of the last context switch to τ_i), and the task's *replenishment queue* R_i (described below). Whenever the task is running, the scheduler increments u_i by the amount of time the task is scheduled, and whenever the task's budget is replenished, u_i is reset or reduced accordingly. Tracking the used budget u_i rather than the remaining budget $\hat{b}_i$ helps with avoiding integer underflow and accounting precision issues if the underlying hardware platform's one-shot timers are not perfectly precise.

Replenishment queue. A task's replenishment queue R_i is a fixed-size circular buffer of tuples, where each entry (t, b) represents an upcoming replenishment that is to grant b units of budget at time t. We let $S_i = |R_i|$ denote R_i's capacity. Initially, the queue's first entry is initialized with the initial budget of the task b_i and the replenishment time set to the task's initial activation time t_{act} (Figure 8a).

We let h_i denote the index of current head of R_i, and for brevity let $t_{head} = R_i[h_i].t$ and $b_{head} = R_i[h_i].b$ denote the current head's timeout and budget values, respectively. The replenishment queue is naturally sorted by time, from soonest to latest replenishment, so that h_i always points to the next replenishment to be processed. When $R_i.dequeue()$ is called, the head h_i is advanced in the usual circular fashion.

SPR uses R_i to track the currently available budget and to plan future replenishments (§IV-B). New replenishments are queued when SPR accounts for τ_i's budget use, either when the task runs out of budget ($u \geq b_{head}$) or when it self-suspends (*e.g.,* due to a blocking call). For example, Figure 8b illustrates a depletion scenario, where τ_i fully exhausted its budget. A new replenishment amounting to the consumed budget b_1 is scheduled for time $t_1 + p_i$, after which h_i is advanced to point to (t_2, b_2). Since the indicated current time t_{now} is earlier than the new $t_{head} = t_2$, the task has no budget remaining and moves to the *depleted* state.

Figure 8c shows an example of SPR batch-processing multiple expired replenishments, which are consolidated into a single budget b_{head} upon activation of the task. Here, three replenishments (b_1, b_2, b_3) expired before the current time t_{now}. The replenishment amounts b_1 and b_2 are added to b_3, and (t_3, b_3) becomes the new head. Since $t_{head} = t_3 \leq t_{now}$, the task has budget to spend and moves to the *ready* state.

When the replenishment queue reaches its maximum capacity S_i, we follow the approach described by Stanovich et al. [21]. Suppose a replenishment (t, b) is to be added to R_i when it is already full, and let (t_{last}, b_{last}) denote the last replenishment in the queue. Then SPR postpones last entry's replenishment time to $t_{last} \leftarrow t$ and updates its replenishment amount to incorporate the additional budget: $b_{last} \leftarrow b_{last} + b$. This avoids undue interference to other tasks, thus maintaining temporal isolation guarantees, but penalizes the task in question by delaying its last replenishment. The reservations of self-suspending real-time tasks should thus be provisioned with a large enough capacity S_i such that R_i never fills completely.

B. Scheduling Logic

The scheduler is triggered by either a timer interrupt or a cooperative call to the scheduler (*e.g.,* to block or activate a task). Each time the scheduler is invoked, it first captures a timestamp (on x86, via the `rdtsc` instruction) that defines the "current time" and forms the basis for all decisions throughout the scheduler invocation (t_{now} in all algorithms and figures). After processing the reason for the invocation (*i.e.,* activation or self-suspension), or in response to a timer interrupt, the scheduler determines the next task to run by calling the *schedule* function. The *schedule* function performs the following activities in order: 1) budget accounting, 2) timeout processing, 3) selection of the next task to execute, and 4) programming of the hardware's one-shot timer.

Budget accounting. The scheduler first processes the previously running task τ_p to account for the budget it consumed and to plan any necessary replenishments, as given in Algorithm 1. Recall that the used budget u_p reflects the task's runtime (w.r.t. t_{now}), which Algorithm 1 assumes to have been updated already. While u_p exceeds b_{head} (line 1), u_p is reduced by b_{head}, a full replenishment of b_{head} is planned for the earliest possible time $t_{head} + p_p$, and h_p is advanced.

To prevent prematurely processing replenishments (where $t_{head} > t_{now}$), the system sets a timer to fire exactly at the time the available budget will be used up (discussed below), ensuring that the scheduling loop does not pull in future budget early. However, if a task completely depletes its budget, there is still a small window of time—from the

```
Algorithm 1: Budget accounting
   /* $\tau_p$, $u_p$, $p_p$: previously executed task, used budget, period
      $R_p$, $(t_{head}, b_{head})$: $\tau_p$'s replenishment queue and its head */
1  while $b_{head} \le u_p$ do
2      $u_p \leftarrow u_p - b_{head}$
3      $R_p.enqueue(b_{head}, t_{head} + p_p)$
4      $R_p.dequeue()$                    /* updates $(t_{head}, b_{head})$ */
5  if $\tau_p.state = blocked$ then
6      if $t_{head} > t_{now}$ then /* blocked and depleted */
7          if $\tau_p.timeout < t_{head}$ then
8              $\tau_p.timeout \leftarrow t_{head}$
9              $timer\_queue.update(\tau_p)$
10     else /* $\tau_p$ is blocked and has budget */
11         $R_p.enqueue(u_p, t_{head} + p_p)$
12         $b_{head} \leftarrow b_{head} - u_p$
13         $u_p \leftarrow 0$ /* split the budget */
14 else /* $\tau_p$ is ready */
15     if $t_{head} > t_{now}$ then /* $\tau_p$ is depleted */
16         $\tau_p.state \leftarrow depleted$
17         $runqueue.remove(\tau_p)$
           /* set the timeout for replenishment */
18         $\tau_p.timeout \leftarrow t_{head}$
19         $timer\_queue.update(\tau_p)$
```

moment the task depletes its budget until the scheduler handles that event—during which the task may consume a few more cycles (*e.g.,* due to the overhead of the timer interrupt itself or in systems with significant interrupt latency or otherwise imprecise timers). Any such overrun must be deducted from future replenishments to avoid inflating the total budget [21]. Specifically, any residual usage in u_p is left as an overrun and is accounted for subsequently (lines 7 and 15 in Algorithm 1).

Next, SPR handles the case where the task blocks. If the task still has remaining budget (line 10), the budget must be split: the *consumed* part u_p is planned to be replenished at time $t_{head} + p_p$, and the *remaining* budget b_{head} is reduced by u_p. Finally, the used budget u_p is reset to zero.

If, in the process of blocking itself, τ_p has *also* depleted its budget (line 7), SPR must prevent a premature activation (§III-C). If τ_p requested a timed wake-up (*e.g.,* via `nanosleep`), $\tau_p.timeout$ holds the intended activation time (otherwise, if τ_p blocks for another reason, $\tau_p.timeout = \infty$). The task can execute only when it *both* activates *and* has budget. Thus, SPR avoids a premature timer by postponing $\tau_p.timeout$ to t_{head} if the requested wake-up time is too early.

If τ_p is still in the *ready* state (line 14), it is either being preempted or a timer interrupt fired to force SPR to engage in bookkeeping. If τ_p's budget is depleted (line 15), the task is moved to the depleted state and removed from the runqueue. Additionally, a timer is added to the timer queue corresponding to τ_p's upcoming replenishment at time t_{head}.

Finally, if τ_p is ready and has budget remaining, it is being preempted and no further management actions are necessary: it simply remains ready and on the runqueue.

Timeout processing. Recall that, at any time, each task has at most one active timeout, either a replenishment or a future activation. SPR defers timer interrupts if they do not lead to immediate context switches (§III), which in turn means that the scheduler can encounter a potentially large number of expired timeouts as part of a single invocation.

Using per-priority timer queues, SPR tracks the next timeout separately for each priority level (Figure 7). To defend against attack A1 (§I), SPR processes at most one such timeout per scheduler activation (§III-E). Specifically, SPR processes a timeout only if it belongs to a task that then becomes the highest-priority ready task (τ_{next} in Figure 7). For example, in Figure 7, SPR processes the expired timeout $\tau_{m3}.timeout$, but *not* the also expired timeout $\tau_{l1}.timeout$, which would activate a lower-priority task. (The higher-priority timeouts $\tau_{h1}.timeout$ and $\tau_{max}.timeout$ are skipped because they have not expired yet.) The correct timeout to process can be found easily with a linear sweep of the top elements of the per-priority min-heaps in order of decreasing priority.

Thus, after processing a single timeout, either:

- a higher-priority task's depleted budget is replenished, moving it to the *ready* state for execution again; or
- a self-suspended higher-priority task waiting for a timed activation is moved to the *ready* state.

In either case, because the processed timeout corresponds to a higher-priority task that is re-added to the runqueue, SPR is guaranteed to then switch to that task.

To emphasize the point: processing any other timers would result in premature processing of events related to lower-priority threads that would not be immediately scheduled anyway. As such, the scheduler defers their processing, thereby avoiding the priority inversion that attack A1 seeks to induce.

Next task selection. After 1) the previously executing task τ_p's budget consumption has been accounted for, potentially removing it from the runqueue, and 2) the timeout of the highest-priority task (if any) to become ready has been processed, thereby adding it to the runqueue, the runqueue is in a consistent state. SPR then first checks whether τ_p is presently in a non-preemptive region: if $\tau_p.state = ready$ and $t_{now} < x_p + \eta_p$, then SPR lets τ_p continue executing irrespective of its priority (*i.e.,* $\tau_n \leftarrow \tau_p$ in this case). Otherwise, SPR simply selects τ_n to be the head of the highest-priority non-empty list in the runqueue (*e.g.,* τ_{m3} in Figure 7).

Hardware timer. After selecting the next task to run, denoted τ_n, the scheduler's final activity is to program the hardware's one-shot timer to regain control at the appropriate time. This time is determined by Algorithm 2 according to three constraints: 1) the amount of budget available to τ_n, including any early replenishments (§III-B), 2) the length of τ_n's non-preemptive region (§III-F), η_n, and 3) the earliest timeout of any higher-priority task, denoted t_{hp}. Lines 1–4 compute the *depletion time* t_d and thus correspond to constraint (1); constraints (2) and (3) are integrated in lines 5–6.

First, the depletion time t_d is initialized to reflect the remaining budget available from the current head of τ_n's replenishment queue, which will be fully consumed after executing for $b_{head} - u_n$ units of budget (line 1). SPR then iterates over every later replenishment in R_n (if any) to identify any budget eligible for early replenishment: if the next

Algorithm 2: Programming the hardware timer

```
/* $\tau_p$, $\tau_n$: the previous and next tasks to be scheduled
   $u_n$, $x_n$: the used budget and last start time of $\tau_n$
   $R_n$, $(t_{head}, b_{head})$: $\tau_n$'s replenishment queue & its head
   $t_d$: depletion time of $\tau_n$
   $t_{hp}$: closest timeout of a higher-priority task */
1 $t_d \leftarrow t_{now} + b_{head} - u_n$
2 foreach $(t, b)$ in $R_n$ after $(t_{head},\ b_{head})$ do
3     if $t_d \geq t$ then
4         $t_d \leftarrow t_d + b$ /* enact early replenishment */
5 if $\tau_p \neq \tau_n$ then $x_n \leftarrow t_{now}$
6 $set\_timer(\min(t_d,\ \max(t_{hp},\ x_n + \eta_n)))$
```

replenishment (t, b) is due by t_d (line 3), its budget amount b can be added to t_d since its replenishment at time t_d is a foregone conclusion (line 4), and so on. Since R_n is ordered by increasing replenishment times, the loop can be aborted when the condition in line 3 is not satisfied.

If the result of the scheduler invocation is a context switch (*i.e.,* $\tau_p \neq \tau_n$), x_n records the time at which τ_n starts executing (line 5), which defines the start of its non-preemptive region. Finally, SPR considers the time t_{hp} at which the next timeout of a higher-priority task is due (as illustrated in Figure 7), where $t_{hp} = \infty$ if no such timeout exists, and the end of τ_n's non-preemptive region at time $x_n + \eta_n$. Task τ_n must be interrupted at latest at time t_d, but potentially already earlier at time t_{hp}. However, if t_{hp} is before $x_n + \eta_n$, the higher-priority timeout must be deferred until the end of the non-preemptive region. Line 6 programs the hardware timer accordingly.

C. Task Activation

A task is activated either by a timer interrupt (*e.g.,* at the end of `nanosleep`) or some other, time-unrelated wake-up event (*e.g.,* when resuming after blocking on a mutex). In either case, SPR updates the activating task τ_a's budget and state with Algorithm 3 before *schedule* is called.

Since SPR does not process replenishments for blocked tasks (§III-C), SPR needs to batch-process any replenishments that expired in the meantime when a task activates. To this end, lines 1–3 of Algorithm 3 accumulate all expired replenishments (if any) into a single one, which becomes the new head of τ_a's replenishment queue R_a, as illustrated in Figure 8c.

Next, SPR checks whether the activating task τ_a has budget available. The simpler scenario is that τ_a is out of budget, $(t_{head} > t_{now})$, in which case it is simply moved to the *depleted* state and a timeout is registered corresponding to the time of its next replenishment (lines 8–11).

Conversely, if τ_a does have budget available (lines 4–7), it is moved to the *ready* state and added to the runqueue. The update of t_{head} in line 5, however, requires some elaboration.

First, recall that $\tau_a.timeout$ holds either (i) τ_a's requested wake-up time, (ii) τ_a's deferred wake-up time (recall line 8 in Algorithm 1), or (iii) $\tau_a.timeout = \infty$ (if τ_a blocked for some timer-unrelated reason). In case (ii), $\tau_a.timeout = t_{head}$, so line 5 in Algorithm 3 has no effect: t_{head} simply remains the replenishment time that τ_a's activation was deferred to.

Algorithm 3: Task activation

```
/* $\tau_a$: the task to be activated
   $R_a$, $(t_{head}, b_{head})$: $\tau_a$'s replenishment queue & its head */
/* Batch-process any expired replenishments, where
     $t_{next} = R_a[(h_a + 1) \mod S_a].t$ and
     $b_{next} = R_a[(h_a + 1) \mod S_a].b$ */
1  while $|R_a| > 1$ and $t_{next} \leq t_{now}$ do
2      $b_{next} \leftarrow b_{next} + b_{head}$
3      $R_a.dequeue()$ /* changes $h_a$, i.e., $(t_{next}, b_{next})$ */
4  if $t_{head} \leq t_{now}$ then /* Is budget available? */
5      $t_{head} \leftarrow \max(\min(t_{now},\ \tau_a.timeout),\ t_{head})$
6      $\tau_a.state \leftarrow ready$
7      $runqueue.add(\tau_a)$
8  else /* $\tau_a$ has no budget available */
9      $\tau_a.state \leftarrow depleted$
       /* Set the timeout for replenishment. */
10     $\tau_a.timeout \leftarrow t_{head}$
11     $timer\_queue.update(\tau_a)$
```

In cases (i) and (iii), since $\tau_a.timeout \leq t_{now}$ if and only if $\tau_a.timeout$ holds τ_a's requested wake-up time, the inner min selects $\tau_a.timeout$ if it is defined, and defaults to t_{now} otherwise. Here, t_{head} needs to be updated to account for the fact τ_a was inactive past its earliest-possible replenishment time, so future replenishments need to be planned relative to the current activation, rather than past replenishments [20].

In an ideal, overhead-free system, t_{head} could simply be set to $\min(t_{now}, \tau_a.timeout)$; however, in a system subject to timer interrupt latency or otherwise imprecise timers, the outer max in line 5 protects against a scenario wherein both $t_{head} \leq t_{now}$ and $\tau_a.timeout < t_{head}$, which is possible if a delay in processing τ_a's timeout pushes t_{now} past t_{head}.

Due to space constraints, we have elided additional logic from Algorithm 3 that resets u_a in the corner case of τ_a suspending for longer than one replenishment period after blocking with a fully depleted budget and an overrun $u_a > 0$.

Finally, after Algorithm 3 ends, *schedule* is called (§IV-B).

V. Evaluation

We carried out experiments with two goals: 1) to investigate how traditional implementations of classic real-time reservation policies are susceptible to interference channels that defeat their theoretical temporal isolation guarantees; and 2) to validate that SPR's constant-time design successfully mitigates these shortcomings. In particular, we introduce novel pathological task behaviors that can induce significant interference affecting higher-priority tasks, and that prevent the effective use of processor time by lower-priority reservations. We also replicate the Thundering Herd [15] attack, which provides another possible vector for priority inversion.

Setup. We ran all experiments on a Cincoze DX1200 embedded computer with an Intel Core i9-12900TE CPU with 8 performance cores (P-cores) and 8 efficiency cores (E-cores) running between 0.8 GHz and 1.1 GHz, with 8 GB of RAM. A single P-core was used for all experiments, with hyper-threading disabled. We used the Composite operating

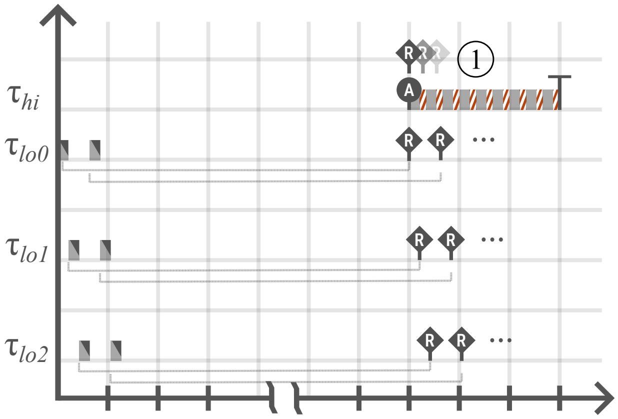


Fig. 9: Illustration of a coordinated timer interference attack delaying a high-priority task via budget replenishments.

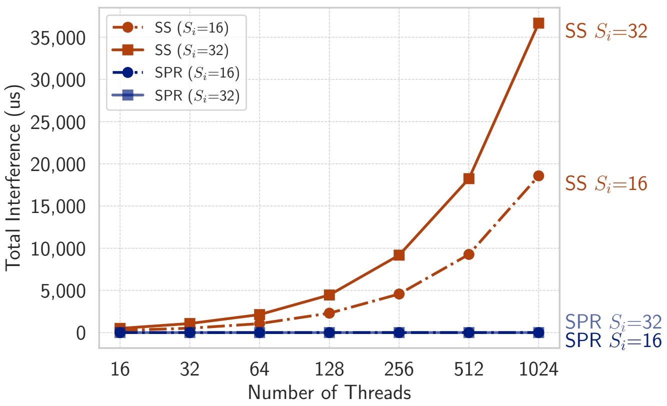


Fig. 10: Interference from timer interrupts due to lower-priority replenishment processing under the SS and SPR policies.

system, in which we implemented three fixed-priority policies: deferrable servers (DS), sporadic servers (SS), and SPR.

A. Mitigation of Coordinated Timer Interference Attacks (A2)

In order to evaluate the effects of unnecessary timer interrupts (as explained in §III-A), we focused on the SS baseline and SPR, as the SS policy's potentially multiple pending replenishments exacerbate the problem. In the experiment, we coordinated replenishments for N (attacking) low-priority tasks $\tau_{lo0}, \tau_{lo1}, \ldots, \tau_{loN}$ to induce unnecessary timer interference in a (victim) higher-priority tasks τ_{hi}.

Let S_i denote the configured replenishment buffer size (*i.e.,* the maximum number of pending replenishments per task). To fill their replenishment buffers, each of the low-priority threads self-suspends (very briefly) S_i times, to be woken again immediately by an even lower-priority auxiliary thread.

We carefully arranged for the low-priority threads to schedule replenishments coinciding with τ_{hi}'s activation time, as shown in Figure 9. In this scenario, under the SS policy, each replenishment causes a separate timer interrupt.

To measure the interference caused by these unnecessary timer interrupts ((1) in Figure 9), we let the high-priority thread τ_{hi} execute the `rdtsc` instruction in a tight loop, recording any samples separated by more cycles than a threshold indicating undisturbed consecutive measurements. The `rdtsc` instruction reads the processor's *time-stamp counter* (TSC), a 64-bit register that counts CPU cycles since reset, providing a fine-grained measure of elapsed time. By detecting and accumulating delays above a set threshold, τ_{hi} can quantify the impact of timer interrupts on its execution.

Figure 10 shows the observed results as a function of the number of adversarial low-priority threads N, under both the SS and SPR policies, in two configurations each for $S_i = 16$ and $S_i = 32$. On the one hand, the effects on the SS policy are clear: As N and S_i increase, the number of timer interrupts and hence the cumulative interference that the high-priority victim thread experiences also increases proportionally.

On the other hand, SPR defers lower-priority replenishments until the high-priority thread finishes its execution, thus preventing excessive priority inversion, which is confirmed by the flat, N-invariant trend of the SPR results.

While Figure 10 covers an admittedly wide range of task counts (up to 1024 threads), evaluating such extreme tasks counts serves primarily to clearly reveal the underlying trend. However, major problems manifest already for moderate thread counts—for instance, even with only 64 threads, delays under the SS policy reach already about 2.5 ms. Given that modern real-time systems in mixed-criticality domains (*e.g.,* automotive, UAVs, and rovers) routinely run 100+ tasks, and that tasks with periods as low as 1 ms are not uncommon (*e.g.,* in automotive systems [10, 11]), delays in the order of several milliseconds from only a few dozens of tasks are concerning.

In summary, our results demonstrate a practical breakdown of temporal isolation under the SS policy when faced with an antagonistic workload. SPR mitigates this attack vector.

Batched replenishment processing. Although SPR's deferred replenishment and activation processing prunes unnecessary timer interrupts, there is a downside to this approach: *batched* processing of *multiple* deferred replenishments before a task is dispatched. Instead of processing replenishments immediately, SPR potentially processes them in batches, which can lead to higher scheduling overhead relative to the baseline SS policy.

To investigate this tradeoff, we measured the execution time of the code for processing early and deferred replenishments in SPR, and report the average and maximum measured overhead in Table II. We collected one million samples for different S_i settings with synthetically filled replenishment buffers.

The results show that the overheads incurred due to batched replenishment processing are not unreasonable, although noticeable for larger S_i settings, and especially so in the worst case (*i.e.,* when considering the maximum). However, these overheads are fully accounted for as part of scheduling and context-switching overhead in O_i^{pre}. Overall, deferred replenishment processing clearly represents a favorable trade-

TABLE II: SPR batch replenishment processing overheads.

S_i	Average Cost ± stdev (cycles)	Maximum Cost (cycles)
1	**52** ± 2	143
4	**61** ± 1	240
16	**95** ± 2	510
32	**159** ± 2	416

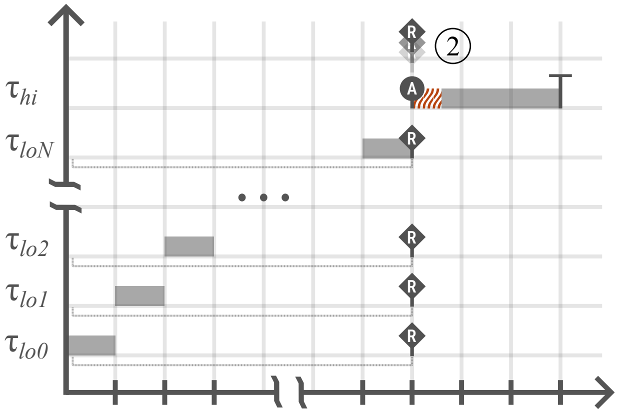


Fig. 11: A Thundering Herd delaying a higher-priority task.

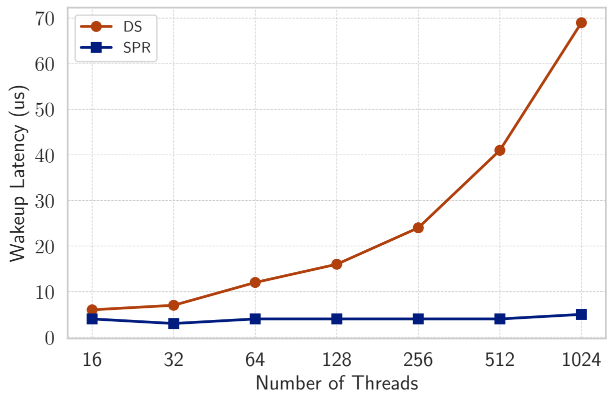


Fig. 12: A Thundering Herd's impact on wakeup latency.

off given the SS policy's vulnerability to coordinated timer interference attacks demonstrated in Figure 10.

B. Mitigation of Simultaneous Replenishment Attacks (A1)

We replicated a variant of the Thundering Herd [15] attack against the DS policy to demonstrate that its simpler replenishment rule is just as vulnerable to exploitation as the more sophisticated SS mechanism. In fact, the DS policy's periodic replenishments make the attack far simpler to implement.

Figure 11 illustrates how the attack proceeds. In general, the replenishment of the N lower-priority tasks are aligned to occur at the same time that a periodic higher-priority victim task is activated. Under the DS policy specifically, the scenario is trivial to set up: the necessary alignment is inherent if the lower-priority tasks share the same replenishment period, and alignment with a higher-priority periodic task's activation time is guaranteed at each common multiple of the victim task's period and the period of the attackers ((2)) in Figure 11).

As a basis for comparison, we implemented a variant of SPR using the DS replenishment rule. To assess the impact of the Thundering Herd (or lack thereof), we measured the *wakeup latency* of the victim task, that is, the difference between its planned activation time and its actual activation time, similar to Linux's `cyclictest`. Figure 12 shows the observed results, again as a function of the number of attacking tasks N.

Under the DS policy, the observed wakeup latency is proportional to the number of lower-priority tasks as eager replenishment processing by the scheduler delays the context switch to the just-activated higher-priority task. In contrast, the delay is constant under SPR as its bounded-time dispatching rule (§III-E) ensures that only a single timeout—here, the one corresponding to the higher-priority task being activated—is processed when the scheduler executes.

Interestingly, Mergendahl et al. [15] reported significantly higher delays than we observed in our setup. We attribute this disparity to algorithmic differences in how reservation timeouts are tracked and processed in seL4 (*i.e.,* the implementation Mergendahl et al. [15] studied) and the fact that we use a much faster and more capable processor in our experiments. Nonetheless, our experiment clearly replicates the trends observed by Mergendahl et al. [15] and thus demonstrates the vulnerability to simultaneous replenishments in DS servers.

In practical systems that commonly aim for wakeup latencies in the single-digit microsecond range, the increases in latency that we observe are by no means negligible. Thread counts in the range of 64 to 128 threads already push the wakeup latency well above 10 μs. Additionally, open systems in which attackers may create additional tasks (subject to admission control) are particularly at risk, as even a very large number of lower-priority tasks can be arranged to have only negligible total utilization (with sufficiently long replenishment periods) and thus virtually no impact on schedulability (§III-G).

C. Mitigation of Preemption Storm Attacks (A3)

To test the efficacy of SPR's non-preemptive regions (§III-F), we used a simple setup: a lower-priority task τ_l, confined to a reservation with $b_l = 1$ s and $p_l = 5$ s, is continuously executing matrix multiplication kernel while a higher-priority task τ_h preempts it potentially repeatedly. As a metric of meaningful progress, we counted the number of matrix multiplications completed by τ_l per replenishment period.

Table III shows the median results across 10 runs for three different behaviors of τ_h and three different settings of η_l. The first row shows a scenario with **no interference**, in which τ_h does not activate. This is the baseline showing τ_l running in isolation (within its configured budget, *i.e.,* subject to regular rate-limiting but without additional preemptions). As expected, the choice of η_l has no impact in this setting.

The second row shows the impact of a **timer flood** caused by τ_h repeatedly self-suspending (with the equivalent of `nanosleep`) for 10μs, causing a deluge of timer interrupts that each "steal" a small amount of execution time from τ_l's budget for interrupt processing and context-switching. Without a non-preemptive region ($\eta_l = 0$), τ_l's throughput is severely reduced to 9650 completed iterations, a 41% reduction relative to the no-interference baseline. In contrast, a 1-ms non-preemptive region restores τ_l's throughput already to 95.5% of the baseline, and with $\eta_l = 2$ ms, τ_l manages to achieve 99.6% of the baseline throughput.

Finally, in the **cache attack** scenario, τ_h causes a timer flood as before, but additionally also flushes the processor's data caches before it suspends, to maximally disrupt τ_l's cache affinity. With $\eta_l = 0$, this almost halves τ_l's throughput, resulting in a 46% reduction w.r.t. the no-interference baseline.

TABLE III: Median number of completed matrix multiplications across 10 runs with and without preemption storms.

Workload	$\eta_l = 0$	$\eta_l = 1\,\text{ms}$	$\eta_l = 2\,\text{ms}$
no interference	16194	16194	16194
timer flood	9650	15458	16125
cache attack	8853	17042	16975

Curiously, with 1-ms and 2-ms non-preemptive regions, we observe marginally *higher* throughput than in the baseline scenario. We attribute this counterintuitive result to a peculiarity in our experimental setup. In particular, we observe higher throughput *with* cache flushes because, after τ_h flushes the cache, τ_l requires only (less costly) cache-line *refills*—which evict unmodified or invalid cache lines—rather than (more time-consuming) dirty-line *write-backs*. This effectively transfers some write-back overhead to τ_h's budget, so that τ_l actually benefits from *infrequent* preemptions by τ_h.

Overall, Table III demonstrates SPR's non-preemptive regions to be an effective mitigation against preemption storms.

D. Transferability and Broader Applicability

While our SPR prototype is based on Composite OS and evaluated on an Intel platform, the underlying insights are not tied to Composite or Intel-exclusive hardware features. Prior research has shown that vulnerabilities similar to those addressed by SPR exist across multiple platforms, including in seL4 on ARM [15] and in Linux on both Intel and ARM architectures [18]. This highlights the broader relevance of SPR's design beyond its current implementation context.

The key principles—pruning unnecessary interrupts, processing at most one timeout per scheduler invocation, and non-preemptive regions—can be seamlessly transferred to monolithic kernels like Linux, microkernels like seL4, hypervisors (*e.g.,* RT-Xen), and containerized environments.

Beyond fixed-priority scheduling, SPR's techniques can be naturally adapted to dynamic-priority systems, such as EDF-based schedulers and in particular (hard) *constant-bandwidth servers* (CBS) [2, 4], by tracking the next-eligible reservation (*e.g.,* by deadline in `SCHED_DEADLINE` [12]) to identify non-essential replenishments that can be safely deferred and to ensure bounded reservation processing overhead per scheduler invocation. Additionally, while we present SPR in the context of single-task reservations for simplicity, multi-task servers can be supported without major logic changes. We leave the practical exploration of such extensions to future work.

VI. Related Work

Reservations have long been used as an OS primitive to control the rate of execution of one or more threads. RT-Mach's processor capacity reserves [14], resource containers [3], and the Resource Kernel [17] all provide resource abstractions to rate-limit arbitrary computations. Modern Linux's `cgroups` [1], widely used as a critical foundation for container infrastructures, are conceptually a direct descendent of this line of work. Similarly, seL4's MCS variant [13] and TCaps [9] provide principled μ-kernel abstractions for reservations, the former focusing on cross-thread communication, and the latter focusing on predictably coordinating multiple cooperating schedulers. SPR's focus is orthogonal to such prior work on principled OS interfaces, programming models, and abstractions, concentrating instead on implementation shortcomings.

As outlined in §I, existing reservation systems suffer from practical limitations that can compromise the isolation guarantees they aim to provide. Stanovich et al. [21] identified issues in the POSIX sporadic server specification itself, demonstrating how these flaws can lead to reservation violations.

The inherent complexity of scheduler implementations, often exacerbated by simple (but in the worst case inefficient) data structures like linked lists, can introduce significant operational overhead. This complexity, coupled with the abstraction provided by budget handling, creates opportunities for malicious tasks to exert undue influence. Mergendahl et al. [15] exposed the Thundering Herd attack against the seL4 sporadic server implementation, which we replicate as attack A1 in §V-B.

Beyond implementation issues, malicious tasks can exploit unrestricted timer APIs to disrupt the execution of other tasks. Patel et al. [18] showed that malicious tasks can use time-based self-suspension operations in Linux (*e.g.,* `clock_nanosleep`) to negatively affect the response times of higher-priority tasks (similarly to our attack A2 in §V-A) and proposed TimerShield, a scheduler-integrated solution to safely defer lower-priority timers. SPR incorporates some of these ideas at the conceptual level, but uses different implementation strategies.

To the best of our knowledge, prior work has not yet explored the difference between reservations for (theoretically fungible) processor time and a task's ability to make meaningful progress in practice (as in our attack A3 in §V-C). SPR is the first comprehensive and principled solution to mitigating all three attack vectors A1–A3, and thus the first reservation scheme that ensures a bound L_i on overhead-induced budget loss (§III-G) in practice, even in the face of malicious tasks.

VII. Conclusions

In this paper, we have demonstrated that despite the strong theoretical properties of established reservation algorithms, their existing direct implementations suffer from a number of practical shortcomings. In particular, we have demonstrated how determined attackers can **(A1)** trigger *simultaneous replenishment* attacks that undermine temporal isolation via induced priority inversion, **(A2)** launch *coordinated timer interference* attacks that similarly result in priority inversion, and **(A3)** cause *preemption storms* that undermine temporal isolation by preventing tasks from making effective use of their guaranteed processor time budgets. In these attacks, the reservation policy's implementation itself is used as an attack surface to undermine its theoretical guarantees.

In response, we have introduced *shielded processor reservation* (SPR) scheduling, the first reservation scheme and implementation blueprint that comprehensively mitigates these shortcomings in practice. To avoid attacks A1–A3 by design, SPR uses early and deferred replenishment processing, prunes unnecessary timers, processes at most two reservations per invocation, and integrates non-preemptive regions. We have shown SPR scheduling to be effective and efficient with the evaluation of a prototype implementation in Composite OS.

Acknowledgements

We'd like to thank our reviewers for their time and effort, which significantly improved this paper. This work is supported by NSF CPS 1837382 and ONR N000142212084. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

References

[1] *Control groups (cgroups)*, 2024. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html

[2] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, 1998, pp. 3–13.

[3] G. Banga, P. Druschel, and J. C. Mogul, "Resource containers: A new facility for resource management in server systems," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, 1999, pp. 45–58.

[4] A. Biondi, A. Melani, and M. Bertogna, "Hard constant bandwidth server: Comprehensive formulation and critical scenarios," in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2014, pp. 29–37.

[5] S. Bozhko and B. B. Brandenburg, "Abstract response-time analysis: A formal foundation for the busy-window principle," in *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2020, pp. 22:1–22:24.

[6] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. A survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, 2013.

[7] "The Composite component-based system source: https://github.com/gparmer/Composite."

[8] K. Elphinstone and G. Heiser, "From L3 to seL4 – What have we learnt in 20 years of L4 microkernels?" in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 133–150.

[9] P. K. Gadepalli, R. Gifford, L. Baier, M. Kelly, and G. Parmer, "Temporal capabilities: Access control for time," in *Proceedings of the 38th IEEE Real-Time Systems Symposium (RTSS)*, 2017, pp. 56–67.

[10] A. Hamann, D. Dasar, S. Kramer, M. Pressler, F. Wurst, and D. Ziegenbein, "WATERS industrial challenge 2017," in *Proceedings of the 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2017.

[11] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," in *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[12] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli, "Deadline scheduling in the linux kernel," *Software: Practice and Experience*, vol. 46, no. 6, pp. 821–839, 2016.

[13] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-context capabilities: A principled, light-weight operating-system mechanism for managing time," in *Proceedings of the 13th EuroSys Conference (EuroSys)*, 2018, pp. 1–16.

[14] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: Operating system support for multimedia applications," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, 1994, pp. 90–99.

[15] S. Mergendahl, S. Jero, B. C. Ward, J. Furgala, G. Parmer, and R. Skowyra, "The Thundering Herd: Amplifying kernel interference to attack response times," in *Proceedings of the IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 95–107.

[16] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.

[17] S. Oikawa and R. Rajkumar, "Portable RK: A portable resource kernel for guaranteed and enforced timing behavior," in *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1999, pp. 111–120.

[18] P. Patel, M. Vanga, and B. B. Brandenburg, "TimerShield: Protecting high-priority tasks from low-priority timer interference," in *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 3–12.

[19] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction.* Berlin, Heidelberg: Springer-Verlag, 1985.

[20] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems*, vol. 1, pp. 27–60, 1989.

[21] M. Stanovich, T. P. Baker, A.-I. Wang, and M. G. Harbour, "Defects of the POSIX sporadic server and how to correct them," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010, pp. 35–45.

[22] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers*, vol. 44, no. 1, pp. 73–91, 1995.

[23] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer, "Speck: A kernel for scalable predictability," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015, pp. 121–132.

[24] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT)*, 2011, pp. 39–48.